A generic FPGA based associative processor

I.Petrescu*. D.Popescu.**
A.Petrescu***

**Quadrilogic Ltd;* (*e-mail:iacobp@quadrilogic.ro*)
***Computer Science Dept., University Politehnica Bucharest*
(*e-mail:dragos@cs.pub.ro*)
**** Computer Science Dept., University Politehnica Bucharest*
(*e-mail:adrian.petrescu@cs.pub.ro)*

Abstract: Field Programmable Gates Arrays (FPFA) enabled the advent of a new computing paradigm, based on direct hardware computational algorithms implementation. The availability of the adequate CAD platforms, together with powerful FPGA's hardware resources, facilitates experimentations with various computational structures. The paper presents a generic, FPGA based, associative computational structure, which may generate implementations of many different applications specific processors, viewed as standalone black boxes, with dedicated inputs and outputs, and able to execute some given simple or complex algorithms. The associative generic processor is parameterized as word-length and word-numbers and operates in a word-parallel/bit-serial mode. Architectural details of the multi-comparand/multi-respondent associative generic processor and its functions are also provided, as well as a number of real algorithm implementations: max, min, sort, select, etc.

1. INTRODUCTION

Data storing and data searching are among the basic operations in computer science and computer engineering. Searching problems arise in different important areas because of the exponentially grows of the stored data. New solutions, based on massively parallel processing, are required in order to speedup search operations. Particularly parallel associative processors are well suited to cope with such tasks. An associative processor comprises, as a basic component, an associative memory in which the words are stored and retrieved in relation to their contents, not on their addresses [***2002], [Foster 1976], [Makimoto 2000], [Hartenstein.,a, 2007], [Petrescu., *et all.*,a 2007]. In order to possess such a feature an associative memory must provide at least the following functions: broadcast of search argument/comparand to all locations, comparison of the search argument-comparand with the content of all locations, identification of matching words and, if necessary, prioritizing multiple matching words. The presence of some facilities for a multi-comparand search, with multi-responders, as well as some logic processing with responder arguments is highly desirable. Associative machines belong to a broader category of parallel SIMD (Single Instruction Multiple Data) machines that are well suited for fast parallel search operations. In the last years, an important progress was achieved, concerning the implementation of a new associative machines with: reconfigurable processing elements interconnection network for embedded applications [Wang.,*et all*,a.2005], associative processor for database applications and image processing [Wang., *et all*,b 2004], scalable ASC processor [Walker., *et all* 2003], associative search and responder resolution features [Wu., *et all* 2002], multi-comparand, multisearch FPGA based associative processors [Kokosinski.,*et all*. a. 2002], etc. One of the main disadvantage of associative memory is the lack of appropriate software. Running existing code on a processor with an associative memory will probably not generate a performance improvement. In order to overcome this problem, completely new algorithms are proposed.

The paper suggests, for algorithm mechanization, a different approach, consisting of a departure from von Neumann programmable machines. In principle, the approach consist of the algorithm depiction by means of a Hardware Description Language (HDL), as Verilog, VHDL, etc., languages. Such a description benefits from the existing powerful CAD platforms, which is assisting the designer in operations like: algorithm checking, algorithm description correctness and, finally, algorithm implementations, by generating a specific configuration stream, for a particular FPGA chip. As it was shown in [Petrescu., *et all.*,b. 2008], FPGA technology enables the implementation of large associative arrays consisting of memory cells that possess, at the bit level, the necessary logic for completion of various associative functions, whose arguments are the associative array, comparands array and masks array. Another component of an application specific associative processor is represented by the respondents array in conjunction with a set of logical functions. The sources of the arguments of these logical functions, as well as the destination of the result, are specified locations of the respondents array.

The connected relationships/functions with the associative array and with the respondents array are specified by the HDL used for a given algorithm description. These relationships/functions are implemented during structural-functional configuration process of the FPGA circuit as a morfware [Hartenstein.,b, 2007].

2. A GENERIC ASSOCIATIVE PROCESSOR.

The architecture of a multicomparand associative processor "seen" by the designer, may be described in terms of a following n-tuple **A**:

$$\mathbf{A} = < \mathbf{IP}, \mathbf{OP}, \mathbf{AA}, \mathbf{CA}, \mathbf{MA}, \mathbf{RA}, \mathbf{R}, \mathbf{O} > \quad (1)$$

Where the following notations are made:

- ✓ IP – input ports set;
- ✓ OP – output ports set;
- ✓ AA – associative array;
- ✓ CA – comparands array;
- ✓ MA – masks array;
- ✓ RA – respondents array;
- ✓ R – AA relationships/functions set;
- ✓ O – RA logical functions set.

The diagram of the proposed generic associative processor. is represented in Fig.1.

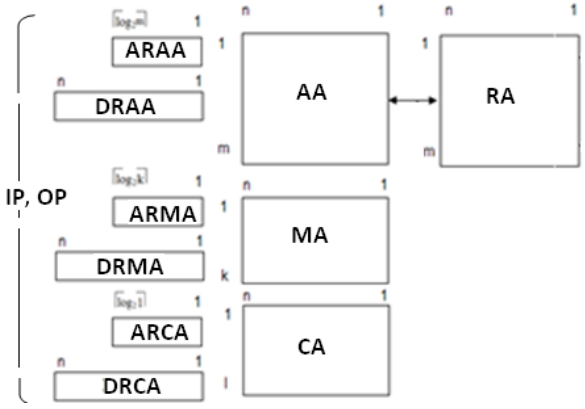

Fig. 1. . The generic associative processor.

Arrays: AA, CA, MA may be seen as memories having *m*, *k*, *l* *n*-bits words. Addressing is done at the word level. When the *j* word is specified, the AA[*j*] notation will be used. The *i* bit from the same word AA[*j*] will be indicate as AA[*j*,*i*]. Similar to RAM operations, the arrays contents are loaded and stored. For these reasons addresses and data registers were provided, specified as ARxx and DRxx, where the suffix xx will be replaced with arrays names: AA, MA, CA.

According to [Hill. *et all*, 1978] the load and store operations may be written as in (2) and (3):

load: DRAA ← BUSFN(AA, DCD(ARAA)) (2)

store: AA*DCD(ARAA) ← DRAA (3)

The multicomparand associative memory AA represents the processor main component [Kokosinski. B. 1997]. At the AA level, data processing is organised in a word parallel- bit serial manner. While data processing is taking place the respondents array RA is loaded with the Cartesian product of data set, stored in AA, and the masked comparand set. The logic attached to each memory cell from AA will enable search operations according to an quintuple (4) of relationships R:

$$\mathbf{R} = \{ <, \leq, =, \geq, >, \} \quad (4)$$

The relationships (4) could be, also, found as relationships operators in Verilog, the language used for algorithms description, simulation and FPGA implementation.

The search operations, seen as Verilog modules: Associative LeSs (ALS), Associative Less or Equal (ALE), Associative Equal (AEQ), Associative Equal or Greater (AEG), Associative GreaTer (AGT) are based on associative combinational networks [Petrescu., c.2008]. In order to specify an associative ALS combinational multicomparand-masked search on AA one can use the (5) notation:

$$RA = ALS(AA, (CA\&MA)) \quad (5)$$

Having in view the existing logical relationships between the operators mentioned in (4), the number of associative networks could be reduced at three, as follows:

$$AGT(AA, (CA\&MA)) = \sim(ALS(AA, (CA\&MA)) \mid AEQ(AA, (CA\&MA))) \quad (6)$$

$$AEG(AA, (CA\&MA)) = (AGT(AA, (CA\&MA)) \mid AEQ(AA, (CA\&MA))) \quad (7)$$

$$ALE(AA, (CA\&MA)) = (ALS(AA, (CA\&MA)) \mid AEQ(AA, (CA\&MA))) \quad (8)$$

The respondents array RA may be visualized as a set of *n* words x *m* bits general registers. The responder *i* is referred as RA[*i*]. A responder register RA[*i*] may be used, as address register, in order to store in AA an operand from DRAA or from masked CA, as is depicted in (9) and (10):

$$AA*RA[i] \leftarrow DRAA \quad (9)$$

$$AA*RA[i] \leftarrow CA\&MA \quad (10)$$

The respondents array RA is associated with an 7-tuple O of logical operators, available, as well as in Verilog language:

$$O = \{ \sim, \&, |, \wedge, \sim\&, \sim|, \sim\wedge \} \quad (11)$$

It is assumed that the respondents processing has 1, 2 or more source operands and only one destination for the result, according to the expression (12):

$$TR[i] = TR[j]\ O_1\ TR[k]\ O_{2....}\ O_{n-1}\ TR[l]\ O_n\ TR[m] \quad (12)$$

where O_i represents one of logical operators from (11).

The next paragraph contains several examples of associative algorithms described in HDL-Verilog, in order to be implemented on specific associative processors suggested by the above discussed generic associative processor. The

proposed algorithms were tested by simulation and implemented in FPGA circuits

3. ASSOCIATIVE ALGORITHMS FOR FPGA ASSOCIATIVE PROCESSORS.

The algorithms that follows (*max, min, sort, select*) assume that the processed data are represented as strict positive integers. The algorithm description in HDL must be preceded by an analyze, in order to establish the range of the representation, precision (bits number) and the displacement to satisfy the operation in the strict positive integers domain.

3.1. max algorithm.

Consider a binary 2D array AA[*m,n*], consisting in *m* rows an *n* columns. As it was mentioned above, this array could be seen as a memory with *m* words/registers/locations x *n* bits. The information in AA is accessed at the word level, so in order to access the *i* bit information from the *j* word one must read the AA[*j*] in an *n* bit P register and latter to refer to the *i* bit from P, namely P[*i*]. If it will be possible to transpose, by hardware, the content of the AA into content of RA, one can operate at the rows level in RA, like at columns as level in AA. Consider the arrays AA, CA, RA expressed in hex and binary (Fig.2.)

```
j=1 AA[j]=8f AA[j]=10001111
j=2 AA[j]=70 AA[j]=01110000
j=3 AA[j]=b8 AA[j]=10111000
j=4 AA[j]=08 AA[j]=00001000
j=5 AA[j]=0c AA[j]=00001100
j=6 AA[j]=e5 AA[j]=11100101
j=7 AA[j]=30 AA[j]=00110000
j=8 AA[j]=90 AA[j]=10010000
j=9 AA[j]=aa AA[j]=10101010
```

AA

```
i=1 CA[i]=80 CA[i]=10000000
i=2 CA[i]=40 CA[i]=01000000
i=3 CA[i]=20 CA[i]=00100000
i=4 CA[i]=10 CA[i]=00010000
i=5 CA[i]=08 CA[i]=00001000
i=6 CA[i]=04 CA[i]=00000100
i=7 CA[i]=02 CA[i]=00000010
i=8 CA[i]=01 CA[i]=00000001
```

CA

```
i=1 RA[i]=14b RA[i]=101001011
i=2 RA[i]=088 RA[i]=010001000
i=3 RA[i]=0cd RA[i]=011001101
i=4 RA[i]=0c6 RA[i]=011000110
i=5 RA[i]=171 RA[i]=101110001
i=6 RA[i]=118 RA[i]=100011000
i=7 RA[i]=101 RA[i]=100000001
i=8 RA[i]=108 RA[i]=100001000
```

RA

Fig.2. The arrays AA, CA, RA expressed in hex and binary.

In order to get the *max* value in AA (Fig.3.), a scanning operation must be done, from left to right, colon by colon, based on the rows CA[*i*] of the comparands array CA- step (1). For each colon, *i,* from AA, an *m* bits respondent RA[*i*] will be generated- step (2). The set of the resulting respondents could be visualized as a binary array with *n* rows and *m* colons, which is the transpose of the AA.
In order to find the *max* value stored in AA a logical *and* operation must be done in RA, line by line: RA[1] , RA[2], retaining the result as a partial logical product.
The partial product is logically *and*-ed with RA[3] and so on. The process will stop when the current result yields a zero bits vector or when the line RA[*n*] is reached. The last nonzero partial product is retained – step (3): 000001000, in the given case. When the product contains a single unit, the index of that unit, counted from left to the right, represents the line number of the *max* value in AA- j=6, step 4.

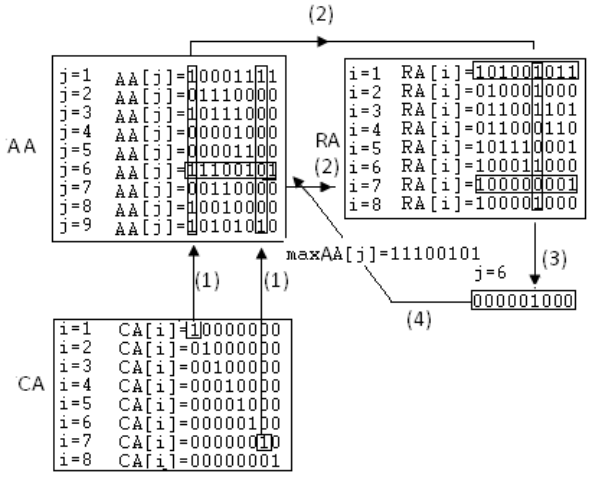

Fig.3.Steps in determining the *max* value in AA.

. One must note that the *max* value in AA is ‘he5, in line 6. The presence of more than one unit in the product denotes that AA possesses a replicate *max* values. The algorithm finds the first *max* value, with smallest index. Fig.4. presents the steps (1) and (2), from Fig.3. which generate RA[*i*].

```
for(i=1; i<n+1;i=i+1)
 begin
  for(j=1; j<m+1;j=j+1)
    begin
      P[j]= ((AA[j]&CA[i])== (CA[i]));
    end
  RA[i]=P;
end
```

Fig.4. Steps (1) and (2) which generate the RA[i].

Consider Rt and Rz to be two *m* bits registers, which represent current and final logical products generated by processing RA lines. The algorithm of the (3) step from Fig.3. is presented in Fig.5

```
for(i=1; i<n+1; i=i+1)
  begin
    Rt=(RA[i]==0)? Rz:(Rz & RA[i]);
    if(Rt==0)
      begin
        Rz=Rz;
      end
      else
       begin
        Rz=Rt;
       end
  end
```

Fig.5. The algorithm for step (3), from Fig. 3.

In order to get (step – 4), the index *j* of the first *max* value, a *casex*( ) Verilog construction was used as in Fig 6. The following program segment displays the index *j*

```
begin
   casex(Rz)
   9'b 000_000_001: begin
          j=9;
         end
   9'b 000_000_01x: begin
          j=8;
. . . . . .
. . . . . .
   9'b 1xx_xxx_xxx: begin
          j=1;
         end
   default:$display(" Continue Rz=%h",Rz);
   endcase
 end
$display(" j=%0d maxTA[j]=%0h  maxTA[j]=%0b ",j, TA[j], TA[j]);
  $stop;
end
```

Fig.6. The program segment which displays the index *j*.

The above discussed *max* algorithm has an O(*m* x *n*) complexity.
The complexity of this algorithms can be reduced by *for* loops parallelization (unrolling), using a Verilog 2001 construction *generate*.

3. 2. min algorithm.

In order to find the *min* value stored in AA, one must replace in the algorithm, which describes the steps (1) and (2), belonging to the *max* algorithm, the AA[j] by ~AA[j], the rest of the algorithm remains unchanged (Fig.7.):

```
for(i=1; i<n+1;i=i+1)
 begin
  for(j=1; j<m+1;j=j+1)
    begin
      P[j]= ((~ AA[j]&CA[i])== (CA[i]));
    end
  RA[i]=P;
end
```

Fig.7. Steps (1) and (2) which generate the RA[i] for *min* algorithm.

The simulation results are presented in Fig.8.

```
j=1 AA[j]=8f AA[j]=10001111
j=2 AA[j]=70 AA[j]=01110000
j=3 AA[j]=b8 AA[j]=10111000
j=4 AA[j]=08 AA[j]=00001000
j=5 AA[j]=0c AA[j]=00001100
j=6 AA[j]=e5 AA[j]=11100101
j=7 AA[j]=30 AA[j]=00110000
j=8 AA[j]=90 AA[j]=10010000
j=9 AA[j]=aa AA[j]=10101010
```

AA

```
i=1 CA[i]=80 CA[i]=10000000
i=2 CA[i]=40 CA[i]=01000000
i=3 CA[i]=20 CA[i]=00100000
i=4 CA[i]=10 CA[i]=00010000
i=5 CA[i]=08 CA[i]=00001000
i=6 CA[i]=04 CA[i]=00000100
i=7 CA[i]=02 CA[i]=00000010
i=8 CA[i]=01 CA[i]=00000001
```

CA

```
i=1 RA[i]=0b4 RA[i]=010110100
i=2 RA[i]=177 RA[i]=101110111
i=3 RA[i]=132 RA[i]=100110010
i=4 RA[i]=139 RA[i]=100111001
i=5 RA[i]=08e RA[i]=010001110
i=6 RA[i]=0e7 RA[i]=011100111
i=7 RA[i]=0fe RA[i]=011111110
i=8 RA[i]=0f7 RA[i]=011110111
```

RA

```
j=4 minAA[j]=8   minAA[j]=1000
```

Fig.8. The simulation result for *min* algorithm.

To the same extent as max algorithm the min algorithm has an O(*m* x *n*) complexity.

3.3. sort algorithm.

The *sort* algorithm is based on repeated (*m* times) execution of the *max* algorithm. On each round *l*, the max value in AA is established. This value is stored in the current location SA[*l*], of an array SA, forcing in 0, in the same time the *max* value in AA. The Verilog code fragment presented in Fig. 9. illustrates the development of the algorithm *sort* based on algorithm *max*:

```
for(l=1; l<m+1;l=l+1)
   begin
      { sort algorithm body }
   end
SA[l]=(i==0)? 0:AA[i];
   TA[i]=0;
  end
  for(l=1; l<m+1;l=l+1)
   begin
    $display("l=%0d SA[l] =%h SA[l]=%b",l, SA[l], SA[l]);
   end
  - - - - - - - - - - - -
```

Fig.9. The development of the algorithm *sort* based on algorithm *max*.

The results of the simulated *sort* algorithm are presented in Fig.10.

```
l=1 SA[l]=e5   SA[l]=11100101
l=2 SA[l]=b8   SA[l]=10111000
l=3 SA[l]=aa   SA[l]=10101010
l=4 SA[l]=90   SA[l]=10010000
l=5 SA[l]=8f   SA[l]=10001111
l=6 SA[l]=70   SA[l]=01110000
l=7 SA[l]=30   SA[l]=00110000
l=8 SA[l]=0c   SA[l]=00001100
l=9 SA[l]=08   SA[l]=00001000
```

Fig.10.The results of the simulated *sort* algorithm.

Since the above discussed *sort* algorithm uses *m* times the algorithm *max*, it has an O(m^2x*n*) complexity.

3.4. sel algorithm.

The *selection* algorithm (*sel*) is used on a large scale, particularly, for accurate surveillance of continuous moving objects and for rapid explorations of some spatial-temporal database systems. With the rapid advances in GPS technologies, it is now become feasible for spatial-temporal database systems to keep track of continuously moving objects accurately. For example, the mobile phone service provider may wish to know how many users are currently present in a specific area. It is believable that the same problem arises in connection with airplanes flying in the neighbourhood of an airport.
Considering a 2D situation, the problem statement is the following. Let $S = \{p_1, p_2, ..., p_n\}$ be a set of moving points in

$\mathbb{R}^2$. For any time *t*, let $p_i(t)$ be the position of p_i at time *t*, and $S(t) = \{ p_1(t), p_2(t), ..., p_n(t)\}$ be the configuration of *S* at time *t*. Given an axis-aligned rectangle $R \subseteq \mathbb{R}^2$ and a time stamp t_q, report $S(t_q) \cap R$, i.e., all points of *S* that lies inside *R* at a time t_q.

In order to illustrate the *sel* algorithm, consider a set of points *S* whose coordinates are represented by pairs of hex digits, for the Y and X coordinates, as in the following file: //CMASOCI_1.txt = (019a, a28b, c30c, 040d, a5ff, 96a0, a7a3, d886, b596). The file could be viewed as an AA content. One may be interested to know what points are placed in a rectangle defined on *x*-axis by coordinates'h86 and 'hff and on *y*-axis by coordinates 'h07 and 'hff. Accordingly the comparands C1, C2, C3 and C4 are defined as:

CA1='h0086; CA2='h00ff; CA3='h0700; CA4='he000;

The selection of X coordinates and Y coordinates is achieved by the following masks:

MA1='h00ff; MA2='hff00;

The *sel* algorithm must generate, at the bit level, the respondents: RA1G, RA1L and RA2G, RA2L for those points from S, which satisfies the above mentioned conditions (Fig.11.):

```
i=1;
  while(i<m+1) // m points number;
     begin
// RA1G respondent bit i generation;
   RA1G[i]=((MA1&CA1)<( MA1&AA[i]));

// RA1L respondent bit i generation;
   RA1L[i]=((MA1&CA2)>( MA1&AA[i]));

// RA2G respondent bit i generation;
   RA2G[i]=((MA2&CA3)<( MA2&AA[i]));

// RA2L respondent bit i generation;
   RA2L[i]=((MA2&CA4)>( MA2&AA[i]));

// Rezult RAz respondent bit i generation;
  RAz[i] = RA1G[i]&RA1L[i]&RA2G[i]&RA2L[i];

     i=i+1;
   end
```

Fig.11. Respondents bit *i* generation.

Because, for *sel* algorithm, it has no relevance, the Verilog code segment, which processes the respondent RAz, in order to display the coordinates of the selected points, is not provided. Fig. 12. contains the results for the above described selection problem.

```
i=1 AA[i]=019a  RA1G[i]=1  RA1L[i]=1  RA2G[i]=0 RA2L[i]=1 RAz[i]=0 Rd=xxxx
i=2 AA[i]=a28b  RA1G[i]=1  RA1L[i]=1  RA2G[i]=1 RA2L[i]=1 RAz[i]=1 Rd=a28b
i=3 AA[i]=c30c  RA1G[i]=0  RA1L[i]=1  RA2G[i]=1 RA2L[i]=1 RAz[i]=0 Rd=xxxx
i=4 AA[i]=040d  RA1G[i]=0  RA1L[i]=1  RA2G[i]=0 RA2L[i]=1 RAz[i]=0 Rd=xxxx
i=5 AA[i]=a5ff  RA1G[i]=1  RA1L[i]=0  RA2G[i]=1 RA2L[i]=1 RAz[i]=0 Rd=xxxx
i=6 AA[i]=96a0  RA1G[i]=1  RA1L[i]=1  RA2G[i]=1 RA2L[i]=1 RAz[i]=1 Rd=96a0
i=7 AA[i]=a7a3  RA1G[i]=1  RA1L[i]=1  RA2G[i]=1 RA2L[i]=1 RAz[i]=1 Rd=a7a3
i=8 AA[i]=d886  RA1G[i]=0  RA1L[i]=1  RA2G[i]=1 RA2L[i]=1 RAz[i]=0 Rd=xxxx
i=9 AA[i]=b596  RA1G[i]=1  RA1L[i]=1  RA2G[i]=1 RA2L[i]=1 RAz[i]=1 Rd=b596
Points number = 4
```

Fig. 12. The selection problem results.

The coordinates of the point of interest are the followings:

{a28b, 96a0, a7a3, b596}

The sel algorithm has an O(*m*) complexity..

3.5. sel algorithm implementation.

To increase the performance of the *sel* algorithm FPGA implementation, the while construction, from Fig.11., was unrolled, using a Verilog 2001 generate operator, as in Fig.12.

```
assign RAz = RA1G&RA1L&RA2G&RA2L;

  genvar i;
  generate
   for (i=1; i <= m; i=i+1)
   begin: slice
   always @ *
     begin
   RA1G[i]=((MA1&CA1)<(AA[(m-i+1)*n:(m-i)*n+1]&
            MA1));
   RA1L[i]=((MA1&CA2)>(AA[(m-i+1)*n:(m-i)*n+1]&
            MA1));
   RA2G[i]=((MA2&CA3)<(AA[(m-i+1)*n:(m-i)*n+1]&
            MA2));
   RA2L[i]=((MA2&CA4)>(AA[(m-i+1)*n:(m-i)*n+1]&
            MA2));
         end
    end
   endgenerate
```

Fig.12. The unrolled respondents bit *i* generation.

The actual implementation was based on the tools provided by Xilinx ISE Design Suite 9.203i. The target device was a FPGA chip: xc2s200e-6pq208, with 200K gates. The experimental platform (Fig. 14.) supported, also, two interfaces: for a PS2 keyboard and for a VGA monitor, in order to input new data, comparands and masks and to visualize the results.

Fig. 13. The experimental stand.

The device report summary, provided by Xilinx ISE Design Suite 9.203i, indicated that the associative *sel* processor implementation used 150 equivalent gate counts. Taking into account as well as PS2 keyboard and VGA monitor interfaces modules, the used equivalent gate count increased to 19, 961.

4. CONCLUSIONS.

The availability, on a relatively large scale, of the FPGA circuits technology, together with powerful CAD tools which facilitate the HDL computational algorithms, description, simulation and implementation greatly influenced the digital systems design. The FPGA based digital systems implementation means: no physical layout process, no mask making, no IC manufacturing , medium costs, tremendous flexibility, etc. In comparison with ASICs, FPGAs decrease NREs(Non Recurrent Expenses) and shortens TTM (Time To Market), [Petrescu, I. d. (2007)].
For the successful and fulness implementation of the new algoritms in FPGA, the user must consider the ways in wich data I/O operations and commands/status are performed: PC host assisted or by means of a dedicated hardware.
The first case is tipical for the development platforms, offering facilities as: versatility, friendly user interface, relatively simple changes of the design, etc. The second case is recommended for stable, embeded solutions in which high data transfer rates are required.

REFERENCES

*** (2002). Associative Memory Study: Architectures and Technology. In:*AFRL_IF_RS_TR_2001_264.Fina Technical Report*. University of Pittsburgh.

Foster, C.C. (1976) Content Addressable Parallel Processors. Van Nostrand Reinhold, New York.

Hartenstein, R. a. (2007). The von Neumann Syndrome. Invited paper. In: *Stamatis Vassiliadis Symposium „The Future of Computing",* Delft, The Netherlands.

Hartenstein, R. b. (2007). Basics of Reconfigurable Computing. Invited paper. In: *DesigningEmbedded Processors. A Low Power Perspective*; J. Henkel, S. Parameswaran (editors): Springer Verlag.

Hill, F.,J., Peterson, G.R. (1978). Digital systems: Hardware organization and design, 2nd edition. John Wiley & Sons. USA.

Makimoto, T. (2000) Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing. In: *The Proceedings of the 10th International Conferenc*e, FPL 2000 August 27–30. Villach, Austria.

Kokosinski, Z.,a, Sikora, W. (2002). An FPGA implementation of a multi-comparand multi-search associative processor. In: *Proc. 12th Int. Conference "Field Programmable Logic and Applications" Lecture Notes in Computer Science*. **Vol. 2438,** pp. 826-835. FPL, Montpellier, France.

Kokosinski, Z.b. (1997). An associative processor for multi-comparand parallel searching and its selected applications. In:*Proceedings of the {IASTED} International Conference on Parallel and Distributed Systems Euro.'97*, June 9-11, 1997. ACTA PRESS. Barcelona, Spain.

Petrescu,I., Nitu, C. a. (2007). Realizarea unei memorii adresabile dupa continut. In:*Revista Romana de Informatica si Automatica.* **vol.17**, **nr**.**4**, ICI. Bucharest.

Petrescu, I., Popescu, D., Petrescu A. b. .(2008). Implementarea algoritmilor de tip CORDIC in FPGAIn: *Revista Romana de Informatica si Automatica,* **vol.18 nr.3**. ICI. Bucharest.

Petrescu, I. c. 2008). An FPGA based associative execution unit**.**In: *The Proceedings of theXXXVIII–International Military Sciences Symposium.* Military Equipment and Technologies Research Agency. Bucharest.

Petrescu, I. d. (2007). Arii/Retele/Tablouri de porti programabile (Field Programmable Gate Arrays –FPGA's). In: *Calculatoare Numerice I*, Cap.7, pag. 157-174. Editura PRINTECH,.Bucuresti.

Sima, D., Fountain, T., Kacsuk, P. (1997), Advanced Computer Architectures: A Design Space Approach, Chapter 12, pp 455-483. Addison Wesley, Harlow, England.

Walker, R. A.,Wang, H. (2003). Implementing a Scalable ASC Processor. In: *Proc. of the 17th International Parallel and Distributed Processing Symposium* (Workshop in Massively Parallel Processing).

Wang,H., Walker, R.A., a.(2005)**.** A Scalable Pipelined Associative SIMD Array with Reconfigurable PE Interconnection Network for Embedded Applications. In: *Proc. of the 17th International Conference on Parallel and Distributed Computing and Systems*, pp. 667-673. Phoenix, Arizona.

Wang, H. Xie, L. Wu, M.Walker, R. A.,b .(2004) A Scalable Associative Processor with Applications in Database and Image ProcessingIn: *Proc. of the 18th International Parallel and Distributed Processing Symposium (Workshop in Massively Parallel Processing)*, abstract on page 259, full text on CDROM. Santa Fe, New Mexico.

.

Wu, M, Walker, R. A., Potter, J. (2002). Implementing Associative Search and Responder Resolution**.** In: *Proc. of the 16th International Parallel and Distributed Processing Symposium* (Workshop in Massively Parallel Processing).

.